Implementation of Multigrid Algorithms on Hypercubes

Tony F. Chan
Ray S. Tuminaro

December, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 86.30

RIACS

Research Institute for Advanced Computer Science

Implementation of Multigrid Algorithms on Hypercubes

Tony F. Chan
Ray S. Tuminaro

December, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 86.30

IMPLEMENTATION OF MULTIGRID ALGORITHMS ON HYPERCUBES†

Tony F. Chan††

University of California, Los Angeles
Research Institute for Advanced Computer Science, NASA Ames

Ray S. Tuminaro†††

Stanford University
Research Institute for Advanced Computer Science, NASA Ames

ABSTRACT

We discuss the implementation of multigrid algorithms for the solution of partial differential equations on a hypercube multiprocessor. We show how the topology of the hypercube fits the data flow of the multigrid algorithm, and therefore allows efficient parallel implementations. We present a timing model for the execution time which predicts accurately experimental results obtained from runs on an Intel iPSC system.

1. Introduction.

The multigrid algorithm is a fast efficient method for solving elliptic partial differential equations on serial computers. The algorithm consists of "solving" a series of problems on a hierarchy of grids with different mesh sizes. For many problems, one can prove that its execution time is asymptotically optimal in that it takes $O(n^2)$ operations to solve the equations corresponding to an n x n grid [11, p 50-51]. No algorithm can do better than $O(n^2)$ since there are n^2 unknowns. Not only is it asymptotically optimal but when properly implemented it is competitive with other algorithms on grids of a modest size [5]. Multigrid is now found in many areas of scientific computation (such as computational fluid dynamics [10]). Given its success on serial computers, it is natural to consider its performance characteristics on parallel machines.

In this paper, we consider an implementation of the basic multigrid method on a distributed memory, message passing hypercube multiprocessor. It would appear that the mapping of the multigrid algorithm to a multiprocessor would be as simple as it is for most other iterative solvers (like the Jacobi method). However, the hierarchy of grids in the multigrid algorithm complicates the flow of data. This in turn may make it difficult to efficiently map the algorithm to parallel machines. We illustrate, however, how the hypercube interconnection topology "naturally" corresponds to the multigrid data flow and thereby makes an

---

†This work is supported by the Research Institute for Advanced Computer Science, NASA Ames, Moffett Field, Ca. and the Department Of Energy under contract DE AC02 81ER 10996.

††Department of Mathematics, University of California - Los Angeles, Los Angeles, Ca. 90024.

†††Department of Computer Science, Stanford, Stanford, Ca. 94305.

efficient parallel implementation possible. Further, it is argued that the multigrid algorithm in a certain sense achieves the optimal execution time for solving an elliptic partial differential equation on a hypercube. Finally, we present a model of the communication and computation for the parallel multigrid algorithm. Using this model, we can predict the performance of the multigrid algorithm on a variety of hypercubes as well as analyze variations of the basic algorithm. We compare this execution model with our computer implementation of the parallel multigrid algorithm on an intel hypercube and find excellent agreement.

2. Jacobi Method.

Before considering parallel multigrid, we briefly outline the Jacobi algorithm which will serve as a typical iterative procedure to compare with multigrid. To simplify the description we restrict our attention to the 1d Poisson equation: $u_{xx} = f(x)$ on $0 \le x \le 1$. The ideas extend naturally to other problems and to higher dimensions. We discretize Poisson's equation by a central difference approximation on a mesh with spacing h, and obtain

$$\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2} = f(x_i) \qquad i = 1, \cdots, n-1 \tag{1}$$

which can be written as

$$Au = h^2 f \tag{2}$$

where A is an $(n-1)$ x $(n-1)$ matrix, and u and f are $n-1$ vectors.
Rewriting equation (1) we get :

$$u(x_i) = [u(x_{i+1}) + u(x_{i-1}) - h^2 f(x_i)]/2 \qquad i = 1, \cdots, n-1 \tag{3}$$

One Jacobi iteration consists of using an approximate solution u to evaluate the righthand sides of equation (3) to obtain a new approximation for each $u(x_i)$.

One nice aspect of the Jacobi algorithm is that it parallelizes easily. If, for example, there are $n-1$ processors, we can put one grid point per processor and evaluate all $n-1$ righthand sides of equation (3) in parallel. In the general case when there are more grid points than processors, each processor is responsible for updating a block of contiguous points. Notice that to apply the formula each processor needs to know the old values of u at the points in its contiguous block as well as the points which border its block. This implies that each processor must communicate with the processors that are assigned to points which are adjacent to its own points. Thus a processor interconnection which topologically matches the difference stencil is sufficient for the parallel Jacobi algorithm. It is well known that when a gray code is used to number contiguous regions, this defines a mapping to the hypercube where adjacent domains are mapped to neighboring processors [4]. Therefore the Jacobi iteration can be made to run with high efficiency on the hypercube.

The major disadvantage of the Jacobi method is its slow rate of convergence. For Poisson's equation it will typically require $O(n^2)$ iterations to converge to the solution for an n x n grid of unknowns. It is this slow rate of convergence that leads us to consider the multigrid algorithm.

3. Multigrid Algorithm.

Only a brief sketch of the multigrid algorithm follows. A detailed description of the algorithm can be found in [1], [7], [8] and [11]. The basic steps are:

1. Apply a couple of iterations of a standard iterative method (for example Jacobi) to produce an approximation : u_1.
2. Set up a system of equations for the error in u_1.
3. "Solve" these equations for the correction on a coarser grid.
4. Interpolate the coarse grid correction and add the correction to u_1 to define the new approximation.

Notice that step 3 involves "solving" a set of equations on a coarser grid for which we can recursively call the same algorithm (ie. use multigrid to solve this coarser set of equations). Below we give the skeleton of a computer code for the basic "V cycle" multigrid algorithm that we have described [8]. The term level is used to denote the grid on which we are currently working on. Level one corresponds to the finest grid. Level two corresponds to the next coarser grid, etc.

```
proc multigrid(f,u,level,pre_relax,post_relax)
{
    if ( level = coarsest level ) then u = (A_level)^-1 h^2 f
    else
        for k = 1 to pre_relax do Jacobi(f,u,level)
        compute_residual(f,u,level,residual)
        project_residual(level,residual,proj_res)
        multigrid(proj_res,v,level+1,pre_relax,post_relax)
        interpolate(level,v,correction)
        u = u + correction
        for k = 1 to post_relax do Jacobi(f,u,level)
    endif
}
```

The main advantage of multigrid is that it converges in a constant (ie. independent of the mesh size) number of iterations and each iteration costs only a constant factor more than that of Jacobi. For large n, this is considerably better than the $O(n^2)$ rate of convergence of the Jacobi method.

Let us consider a parallel implementation of a 1d multigrid algorithm. The basic idea is similar to the Jacobi algorithm. We assign grid points to different processors using a gray code mapping. Specifically, we look at the case when there are $n-1$ processors and $n-1$ unknowns (where $n = 2^k$). We look at this case in detail for two reasons. First, it is less complicated than the case where we have many points per processor and second even if n is greater than the number of processors on the fine grid as we continue to form coarser grids eventually the number of points on the coarse grid will be equal to the number of processors. In this case we assign one point per processor like in the Jacobi algorithm. The coarse grid is defined by taking every other point from the fine grid. Notice this implies that we will have many idle processors on the coarser grids. Let's look at processor $n/2$. To perform the residual projection, interpolation, and Jacobi iterations, this processor has the following communication needs:

finest grid level 0 : communicates with processors $n/2 - 1$ and $n/2 + 1$.
grid level 1 : communicates with processors $n/2 - 2$ and $n/2 + 2$.
grid level i : communicates with processors $n/2 - 2^i$ and $n/2 + 2^i$.

Thus the multigrid algorithm requires more sophisticated communication links than Jacobi. In other words, a simple processor grid which matches the stencil of the partial differential equation is not sufficient for an efficient multigrid algorithm. We state without proof the following result. If a particular gray code (specifically the binary reflected gray code) is used to assign grid points to processors on a hypercube, then the processors that must communicate with each other in the multigrid algorithm are at most a distance of two away from each other (regardless of the level of the grid and the size of the hypercube). Further, there is a simple and efficient algorithm that allows one to shuffle the grid points to different processors before moving to a different level so that we can maintain communication links of a distance one. We omit the details and refer the reader to [3]. The key point is that by properly mapping a problem on a hypercube, our communication needs remain local no matter how coarse the grid is compared to the size of the hypercube.

4. A Lower Bound Based on Data Flow in Solving PDE's.

In this section, we shall introduce a data flow view of algorithms for solving elliptic pde's and use it to derive a lower bound on the execution for solving such equations on hypercube computers.

The solution of an elliptic pde at any point in the domain requires some knowledge of information on the boundary. We know for example that by changing the boundary conditions we change the solution inside the domain at all points. Therefore, when we consider the convergence of numerical methods to the solution of elliptic problems, we can get a lower bound by determining the time it takes for the boundary information to reach all points in the interior. For example, consider the Jacobi method applied to the 1d Poisson equation. It takes $O(n)$ iterations before this information propagates to all the interior points. Thus a lower bound for the convergence of the Jacobi method is $O(n)$. Note that it actually takes $O(n^2)$. From this point of view, we can see why the multigrid algorithm yields such rapid convergence. One multigrid iteration propagates the boundary information to all the points in the interior. Thus the lower bound on the convergence of multigrid is $O(1)$ which is the actual convergence rate.

The principle advantage of the hypercube interconnections is that they allow one to efficiently communicate the boundary information globally. For example, if we have one point per processor and the processor $i_1 i_2 i_3 \cdots i_n$ contains some boundary information, it will take $\log n$ steps to propagate this information to processor $\bar{i}_1 \bar{i}_2 \bar{i}_3 \cdots \bar{i}_n$ (where the overhead bar denotes complement). Thus we can conclude that the optimal asymptotic time for solving an elliptic problem with one point per processor on a hypercube is $O(\log n)$. We shall see in Sec 5 that the multigrid algorithm achieves this optimal time as it takes $O(\log n)$ time to perform one multigrid iteration on a hypercube of size n (see also [2] and [6]).

The need for efficient global communication is not particular to multigrid. For example, any numerical algorithm which requires convergence checking needs to be able to communicate global information. Notice that convergence checking within the multigrid algorithm can be performed with almost no overhead. For example if we use the residuals as a measure of convergence, which are already computed in the multigrid algorithm, the norm of the residual vector on the finest grid can be accumulated at the coarsest level using a tree sum method which can be integrated into the multigrid algorithm. In fact by clever programming, the norm of the fine grid residual can be sent in the same messages which are used to transmit the residuals on the lower levels. This method implies that convergence will be determined after one additional multigrid iteration has been performed.

5. Modeling Communication and Computation.

We model the execution time of the parallel multigrid algorithm on a two dimensional $n \times n$ point grid using a $p \times p$ processor grid. The execution of one multigrid iteration consists of performing the Jacobi sweeps, interpolation, residual projection, and "solving" the coarse grid equation. There are two separate cases in the parallel multigrid implementation which must be analyzed slightly differently. Specifically when $n > p$, each processor has ($n/p \times n/p$) points. Thus when we communicate with our nearest neighbor we send messages of length n/p. On the other hand when $n < p$ we have some idle processors and those processors which are not idle contain only one point. So communication with our nearest neighbor requires messages of length one to be sent. Notice that even if $n > p$ on the fine grid, eventually on some level (ie. on some coarse grid) n will be less than p.

We define the following notation:

$T(n)$: time to perform one multigrid iteration on an $n \times n$ grid using p^2 processors.

$\alpha + \beta n$: time to communicate a message of length n between neighboring processors.

t : time to compute one Jacobi sweep at one point on the grid.

ν : total number of Jacobi sweeps that are performed on each multigrid level (ie. pre_relax + post_relax).

r : time to compute the residual at one point.
ρ : time to project one point of the residual onto the coarse grid.
i : time to interpolate from the coarse grid and apply the correction to the previous approximation.
$M = \nu t + r + \rho + i$. The computation time on one level for one point.

We assume in this analysis that the hypercube has bi-directional simultaneous send and receive. If we count the arithmetic operations for the case $n > p$, we have:

Jacobi sweeps : $\nu(t(n/p)^2 + 4(\alpha + \beta(n/p)))$.
- receive information on all four boundaries and compute new approximation at all $(n/p)^2$ points.

compute residual : $r(n/p)^2 + 4(\alpha + \beta(n/p))$.
- receive information on all four boundaries and compute residual at all $(n/p)^2$ points.

project residual : $\rho(n/p)^2 + 4(\alpha + \beta(n/p))$.
- receive information on all four boundaries and project residual at all $(n/p)^2$ points.

interpolate : $i(n/p)^2 + 4(\alpha + \beta(n/2p)) + 4(\alpha + \beta)$.
- receive information on all four boundaries as well as information on the corners to interpolate the correction at all $(n/p)^2$ points.

We can now combine these to obtain a recurrence relation for the execution time of the multigrid algorithm. For $n > p$

$$T_1(n) = T_1(n/2) + M(n/p)^2 + [4\nu + 10]\beta(n/p) + ([4\nu + 16]\alpha + 4\beta).$$

For $n = p$ we have the initial condition

$$T_1(p) = T_2(p)$$

Doing a similar analysis for the case $n \le p$ (using the Chan-Saad shuffle algorithm) we get :

$$T_2(n) = T_2(n/2) + M + [20 + 4\nu](\alpha + \beta)$$

with initial condition

$$T_2(2) = C_1$$

where C_1 is the time to solve the system corresponding to a 2 x 2 grid with one point per processor (and the processors are nearest neighbors). Note these formulas are only valid for $p > 2$ as the assumptions of sending and receiving on four boundaries are not valid for smaller systems. To analyze smaller systems we must modify our assumptions. Solving the above recurrence relations we get

$$T_1(n) = (4/3)M[(n/p)^2 - 1] + d_1[(n/p) - 1] + d_2\log(n/p) + T_2(p)$$

and

$$T_2(p) = d_3\log(p/2) + C_1$$

where

$$d_1 = (8\nu + 20)\beta, \quad d_2 = (4\nu + 16)\alpha + 4\beta, \quad d_3 = M + (20 + 4\nu)(\alpha + \beta).$$

When the ratio n/p is large, the first term dominates and so $T_1(n) \approx (4/3)M(n/p)^2$. Thus when the number of points per processor is large, the execution time is reduced by almost p^2 which is in fact the maximum attainable speed up on a p^2 node hypercube. Considering the optimal nature of serial multigrid, parallel multigrid can also be considered asymptotically optimal $O((n/p)^2)$. At the other extreme when n is large and $n = p$ (ie. one point per processor), then $T_1(n) = O(\log n)$. Our previous discussion concluded that $O(\log n)$ is the optimal execution time for solving elliptic partial differential equations on a hypercube with one point per processor. Thus multigrid is also asymptotically optimal when there is one point per processor and a large number of processors. It is interesting to note that this optimal behavior is achieved even with the many idle processors that result when "solving" on the coarse grids.

The results of the preceding paragraph are encouraging, but they do not indicate the performance of the parallel multigrid algorithm on practical grid sizes and realistic hypercubes. Toward this end, we can use the execution model to predict the actual execution of the parallel multigrid algorithm under different assumptions. Figure 1 shows the predicted runtimes for one multigrid iteration using the machine parameters for both the intel iPSC and the Caltech Mark II hypercube (using sixteen nodes) for different grid sizes. The values for α and β were obtained from data in [9]. For the Caltech Mark- II machine $\alpha = 8.8 \times 10^{-4}$ and $\beta = 8.4 \times 10^{-4}$. The efficiency plots shown in figure 2 indicate how large the ratio (n/p) must be before we are close to the maximum attainable speedup. For the Caltech machine, we see that for $n/p \approx 16$ we reach 80 percent efficiency. This is an indication that the ratio n/p does not have to be large before we get almost p speed up.

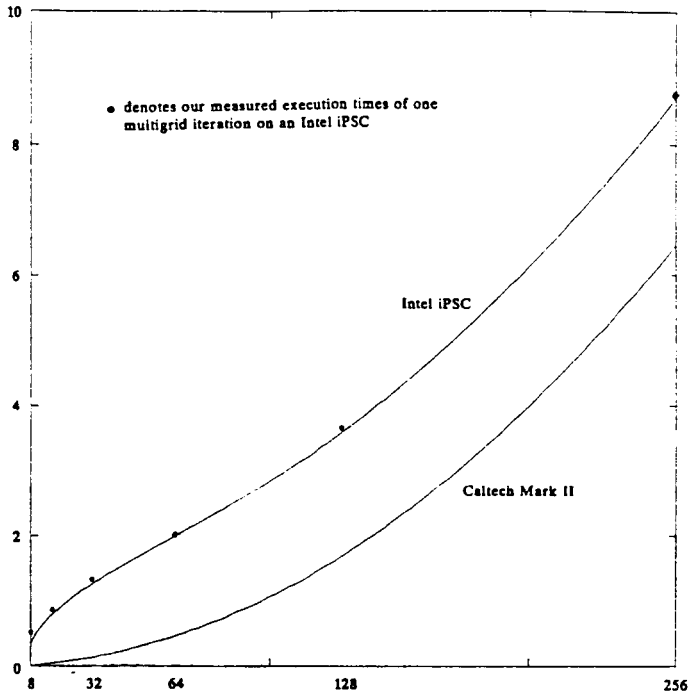

figure 1 : predicted execution time of one multigrid iteration vs. n using 16 processors for grids of size n x n

6. Numerical Experiments.

A computer code of the parallel multigrid algorithm was implemented on the intel iPSC hypercube. This code was used to solve Poisson's equation ($u_{xx} + u_{yy} = f(x,y)$) on a square grid. The Dirichlet boundary conditions as well as the function $f(x,y)$ were chosen so that the exact solution was $u(x,y) = x^2 + y^2$. In the current version of the multigrid code there is no convergence checking. Timing experiments of the parallel multigrid algorithm were run using four nodes as well as using sixteen nodes. The processors were assigned to subdomains using the binary reflected gray code (in the x and y directions). The execution runtimes for one multigrid iteration (averaged over a sequence of iterations) of this code on grids of various sizes is shown in figure 1. The close correspondence between the actual runtimes and the predicted runtimes is an indication that the execution time model accurately reflects the runtimes of the parallel multigrid algorithm.

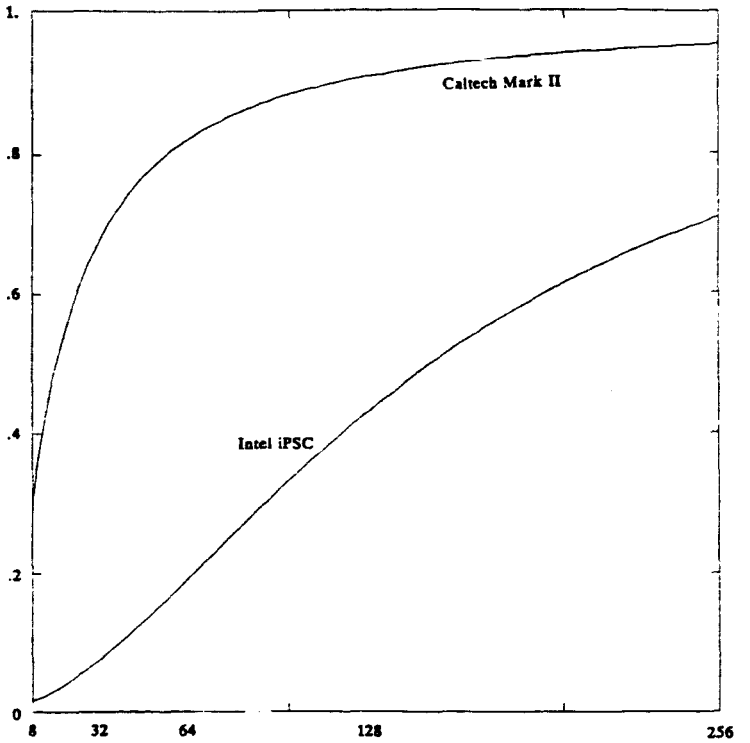

figure 2 : predicted efficiency of one multigrid iteration vs. n using 16 processors for grids of size n x n

7. Conclusion.

It is well known that the multigrid algorithm is among the most effective methods for solving elliptic partial differential equations on serial computers. In this paper, we have shown that it can be effectively mapped to a hypercube so as to maintain its optimal properties. When there are many grid points compared to the number of processors, it is possible to attain almost the maximum possible speed. When there are many processors and only a few points per processor, the multigrid algorithm is also optimal. Specifically, if the ratio of points per processor is fixed at one and the number of processors p is varied, the multigrid algorithm achieves the asymptotically lower bound, $O(\log p)$ for solving pde's. This implies that for large processor systems multigrid is optimal and that for small processor systems where there are many points per processor, multigrid is still optimal.

To determine estimates of execution times on realistic machines and realistic size problems, we presented a model of the communication/computation of the multigrid algorithm. The accuracy of the model was verified by a comparison with the timing results of our multigrid implementation on an Intel iPSC 32 node system. Using the model, it is possible to predict the execution time of the multigrid algorithm on various hypercubes (ie. with different machine parameters). In addition, the model can be used to compare the execution time of different variants of the algorithm. Our preliminary analysis, illustrates that a parallel multigrid can be efficiently mapped to a hypercube and therefore execute significantly faster than on a serial machine.

References

[1] A. Brandt, *Multi-level Adaptive Solutions to boundary-value problems*. Math Comp 31 (1977) 333-390.

[2] A. Brandt, *Multi-grid Solvers on Parallel Computers*, Technical Report 80-23, ICASE, NASA Langley Research Center, Hampton, VA, 1980.

[3] T. Chan, and Y. Saad, *Multigrid Algorithms on the Hypercube Multiprocessor*, IEEE Trans. Comp. Vol. C-35, No. 11, Nov 1986, pp969-977.

[4] T. Chan, Y. Saad, and M. Schultz, *Solving Elliptic Partial Differential Equations on Hypercubes*. In: Proceedings of the First Conference on Hypercube Multiprocessors, M. Heath (ed), SIAM, Knoxville, Tennessee, August, 1985, pp 196-210.

[5] T. Chan, and F. Saied, *A Comparison of Elliptic Solvers for General Two-Dimensional Regions*, Siam J. Sci. Stat. Comput., July 1985, Vol 6, No 3. pp 742-760.

[6] T. Chan, and R. Schreiber, *Parallel Networks For Multi-grid Algorithms: Architecture and Complexity*, Siam J. Sci. Stat. Comput., July 1985, Vol 6, No. 3. pp 698-711.

[7] W. Hackbusch, *Multi-grid Methods and Applications*, Springer-Verlag,1985, Berlin.

[8] D. Jespersen, *Multigrid Methods for Partial Differential Equations*. In: Studies in Numerical Analysis, G Golub (ed), MAA Studies in Mathematics Vol 24, 1984. pp 270-318.

[9] A. Kolawa, and S. Otto, *Performance of the Mark II and Intel Hypercubes*, Caltech Concurrent Computation Group - report 254. Pasadena, CA, Feb 1986.

[10] R. Peyret, and T. Taylor, *Computational Methods for Fluid Flow*, Springer-Verlag, New York, 1983.

[11] K. Stuben and U. Trottenberg, *Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications*. In: Multigrid Methods, Hackbusch, W. and Trottenberg, U. (eds), Koln-Porz, Nov 1981. Lecure Notes in Math 960. Springer, Berlin 1982.